(2)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1989	FINAL report, 01 Nov 88 thru 31 Oct 89

4. TITLE AND SUBTITLE
EVALUATION METHODOLOGY FOR SOFTWARE ENGINEERING

5. FUNDING NUMBERS
AFOSR-89-0080
61102F 2304/A2

6. AUTHOR(S)
Bruce I. Blum

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20707

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 90-0215

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
Building 410
Bolling AFB, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This is the final report on research intended to investigate the most effective methods for software engineering evaluation. The objective of this work is to identify and evaluate the methods used to measure the impact of changes to the software process. In particular, there is a special interest in the evaluation of benefit improvements when different process models are used. The research has pursued two types of activity. First, evaluation methods used in other disciplines have been reviewed for their utility in software engineering. The long-term goal is to produce a taxonomy of methods with a suggested range of strengths for software engineers. The availability of this unified view would help analysts select the most appropriate evaluation techniques for a given class of task. The second class of activity employed small studies in which evaluation methods could be tested and/or quantifiable concepts could be modeled. Because the research goal is to provide a means to appraise alternative development paradigms, most of the effort was spent on the study of an essential software process model (i.e., a meta-process model) and the evaluation of paradigms that alter the process within that model.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	15
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Evaluation Methodology for Software Engineering
Final Report on Grant No. AFOSR-89-0080

Bruce I. Blum

RMI-89-028
December, 1989

Evaluation Methodology for Software Engineering

Final Report on Grant No. AFOSR-89-0080

Bruce I. Blum

Johns Hopkins University/Applied Physics Laboratory

INTRODUCTION

This is the final report on research intended to investigate the most effective methods for software engineering evaluation*. The objective of this work is to identify and evaluate the methods used to measure the impact of changes to the software process. In particular, there is a special interest in the evaluation of benefit improvements when different process models are used.

The research has pursued two types of activity. First, evaluation methods used in other disciplines have been reviewed for their utility in software engineering. The long-term goal is to produce a taxonomy of methods with a suggested range of strengths for software engineers. The availability of this unified view would help analysts select the most appropriate evaluation techniques for a given class of task. Work on this task during the period of the current grant was limited to general reading; no preliminary taxonomy was prepared.

The second class of activity employed small studies in which evaluation methods could be tested and/or quantifiable concepts could be modeled. Because the research goal is to provide a means to appraise alternative development paradigms, most of the effort was spent on the study of an essential software process model (i.e., a meta-process model) and the evaluation of paradigms that alter the process within that model.

This report is divided into two sections. The first describes the problem as it is interpreted in the context of this research grant. The second section presents the accomplishments of this grant.

---

* The work reported on in this document represents the second year of research supported by AFOSR on this topic. The first year of research was funded under grant AFOSR-87-0219. The research described in this report represents the second year of investigations in what initially was planned as a three-year study. Thus, although this report describes what was accomplished as the result of grant AFOSR-89-0080, not all the issues identified in the grant proposal have been addressed.

THE TECHNICAL APPROACH

Computer science and the application of computers are undergoing revolutionary changes. Traditional development paradigms have new tools to support the software process. Examples include Ada and other languages that apply the principles of abstraction and concurrency management, environments and work stations that integrate graphics and text, and general purpose facilities that allow casual users to satisfy their needs directly. New paradigms also are being produced to offer improvements in quality, cost, and scope. Examples here are the use of artificial intelligence and knowledge-based assistants, the direct execution of specifications with the operational approach, and the application of new techniques such as object oriented programming and conceptual modeling. Finally, there are major changes in the hardware environment. For example, lowered costs eliminate many of the memory and processing speed barriers, new parallel architectures remove the earlier processing bottlenecks, and communications and networking blur the boundaries between individual computers and databases.

Yet with all this improvement, we lack a clear understanding of how to evaluate our progress. In some areas, such as equipment cost per unit of memory or processing time per unit of operation, the change is easily quantified. However, when one sets as a goal the improvement of "the power, quality, reliability, and transportability of computer software and the verification of software, data, structure, and operating systems,"[1] how does one quantify the improvement? Moreover, if one asserts that the improvements result from the use of new processes, methods, tools or environments, then how does one identify and evaluate the contributing factors?

This research addresses these issues. Methodology is the study of methods, and the focus of this investigation is the study of evaluation methods -- used in software engineering and in other scientific disciplines -- as they relate to software development, use and maintenance. The goal of this research is (a) to identify demonstrated techniques that can be applied in software engineering, (b) to establish taxonomies of (1) attributes that can be evaluated and (2) the associated evaluation methods, and (c) to document -- by means of references and pilot studies -- which metrics offer valid measures of improvement and which qualities can be evaluated only subjectively. Naturally, to provide a context for the evaluation, the research also involves a definition of the essential characteristics of the processes to be measured.

Problems in Software Engineering Evaluation

There are two classes of evaluation in software engineering. The first, which we call vertical evaluation, entails evaluation of software within a fixed context. Typical examples of this type entail the collection of data project attribute data (such as product size, changes and failures) to construct models that predict cost, quality, reliability, etc. Such evaluations normally are performed for a fixed development community with a given process model over an extended period of time. During the period of data collection there tend to be changes to the process and environment, and the data analysis is used either to measure improvements or to predict future performance. For example, cost models are based upon empirical evaluations of previously collected data. The vertical evaluations are most valid when they are based on longitudinal data from a single organization. Comparisons across organizations have broader variance, and few industry-wide standards have been accepted.

Horizontal evaluations -- the target area of this research -- focus on the evaluation of technology with respect to its impact on the software process. That technology generally is presented in the form of a tangled hierarchy as follows:[2]

Process Descriptions
|
Methods and Practices
|
Tools
|
Support Environment

Thus, for a fixed process model and set of practices, there are tools that can support that model. There also are alternative approaches for combining these tools to produce support environments for software development and maintenance. The goal of horizontal evaluation, therefore, is to measure the impact of changes at any of the four levels of this software engineering hierarchy.

Horizontal evaluation is difficult. First, computer science is unlike other sciences; its scientific base rests in the formalisms that it uses. These formalisms have logical properties that can be evaluated independent of any application. Indeed, Turski notes,[3]

> The history of advances in programming -- the little that there is of it -- is the history of the successful formalization: by inventing and studying formalism, by extracting rigorous procedures, we progressed from programming in machine code to programming in high level languages (HLLs).... For many application domains HLLs provide an acceptable linguistic level for program (and system) specification.... The expert views of such domains, their descriptive theories, can be easily expressed on the linguistic level of HLL, thus becoming prescriptive theories (specifications) for computer software.

In software engineering the primary object of interest is the software product, not its programming. Research is concerned with tools to support the development of descriptive theories in the problem domain, the transformations and practices necessary to formalize a HLL prescriptive theory that can be implemented as a software product, and the management and support of this process.

The kinds of evaluation appropriate for this research cannot follow the models of evaluation used in physics and engineering. There are no fixed phenomena one cannot test a theory empirically because the data are affected by too many uncontrolled variables. This complexity also makes it difficult to separate the attributes to be evaluated from the background effects. The cost of collecting data is high, and there are difficulties in establishing controls in "real" (as opposed to "toy") projects. In fact, controlled studies with sample sizes large enough to evaluate a hypothesis are possible only for the most constrained issues.

In addition to the problem of not having well defined properties to be evaluated, there are -- as we noted in the discussion of vertical evaluation -- few broadly accepted baselines or "gold standards" against which one can measure change. Software engineering is dynamic, and it is not clear how data collected over a span of two decades can be used. For example, a frequently cited fact is that there can be a 1:28 variation in programmer

performance. However, this is based upon one element in a 12 programmer study conducted in the late 1960s.[4] Is this valid in today's age of personal computers and computer literacy? Was the difference an artifact of training that would correct itself as more effective methods were learned? Recall that the QWERTY keyboard format initially was chosen because it would slow performance and thus prevent the jamming of keys. This justification for its selection no longer is valid, but the keyboard design remains a persistent standard. In this spirit, some older assumptions about the software process should be reexamined.

Finally, there are inherent problems in the evaluation of software engineering technology. First, much of the research and development with respect to technology is either academic or proprietary. Evaluation in an operational environment may not be possible. Second, the academic research is complex and typically requires years to complete. Thus, much that is reported must be descriptive, conceptual and/or subjective. Moreover, much of the software engineering technology that is investigated in a research setting has no parallel in a production setting. (For example, with the current levels of experience, it is not practical to evaluate the methods used to implement an expert system.) There also are unavoidable biases in evaluating a technology. Mahoney has studied the problem of self-deception in science and argues that "the psychological processes powerfully influence and, in many ways, constrain the quality of everyday scientific enquiry."[5] By way of conclusion, we note that all of these problems are further exacerbated when one performs this inquiry in a dynamic discipline, with a limited heritage of formal evaluation, and where there is a strong personal bonding with the objects of study.

A Framework for Evaluation

In establishing a framework for horizontal evaluation, there are two basic approaches. One can start by identifying the objects to be evaluated, or one could begin with methodological issues. We start with the first.

In a software engineering context, there are three objects that can be evaluated:

Problem. This is the application or need for which a software product is being developed.

Process. This is the sequence of activities associated with the software product's development and maintenance.

Product. This is the software item that is delivered and used.

There have been few attempts to fix a problem and investigate alternative approaches with respect to its implementation. Boehm's COCOMO system has been used as a control for some student exercises[6,7] and as a baseline for other studies.[8] Cugini has built a database of programs for a fixed, non-trivial problem, but the data have not been studied in any depth.[9] In each of these cases, a fixed problem was used to evaluate some properties of the process or the product. Halstead, on the other hand, introduced Software Science as a theory for repeatable and universal measures at the problem level.[10] The problems, in this case, were limited to algorithms, and much of the initial theory is no longer accepted as valid. Albrecht sought an alternative approach to quantifying the size of a problem; he introduced the concept of function points as a measure for information processing problems.[11]

(Interestingly, the extensions to function point analysis focus on estimating the size of, and therefore the effort to produce, the end product.)

With respect to the process, there have been many evaluations of the impact of change within the context of a fixed process model. Most of this evaluation is what we already have termed vertical, i.e., it compares effects within a general problem domain and often within a single organization. The analysis of cost and schedule data are examples of this type of evaluation; such data would be useless for comparisons with process models that use tools or paradigms that distort the allocation of effort among the process steps. For example, how does one use historical costing data from traditional production cycles to estimate costs when using a Fourth Generation Language?

Finally, there is the software product. Most evaluations focus on attributes of the product.[12] Some obvious measures are lines of code and numbers of errors encountered. Code often is considered the first formal object that can be analyzed, and there are many easily computed metrics that are used to predict product quality. (McCabe's complexity metric[13] is one commonly applied example.) Nevertheless, virtually all evaluations of a final product hold the process model and environment fixed. Few studies are designed to address the impact of a major technological change, and many of those that do are naive in their study designs. For example, to what extent did the improvement associated with structured programming result from the introduction of discipline, the reduction in size of the conceptual objects being processed, or the Hawthorne effect? Although the question may seem facetious, if the structured approach were accepted because of its side effects, then its rigid retention might become a software parallel to the QWERTY keyboard.

Given this stratification of the problem domain, what evaluation methods can be applied? It is useful to start with a medical model.[14] At the lowest level, there is the basic research in biological phenomena. This involves *in vivo* and *in vitro* studies and the use of mathematical and animal models. The goal is to isolate some portion of a biomedical problem so that it can be understood better. Examples in the computer science domain include evaluation of algorithms and transformations, determination of user reaction responses to different interfaces, and the measurement of some properties of code or documentation. In each of these cases, the objective is to establish some invariants within a given context that add to our understanding of some larger problem.

At the next level are case studies and clinical trials. In medicine, more time is spent in training a specialist in a clinical setting than in a classroom. The amount of formal knowledge available is beyond the comprehension of a single individual; therefore, much of the physician's training is organized around the clinical situations that he is expected to encounter. The result is a set of learning experiences derived from case studies, i.e., specific instances. Experience has shown, however, that we are poor judges of outcome when we generalize from anecdotal records. Thus, medical research confirms its perceptions by the use of clinical trials. Here a cohort (a group of like patients) is selected, a set of procedures or therapies is defined that involves a limited number of variables, and the outcome is used to evaluate some null hypothesis. Most computer science examples of this type of evaluation are rooted in the behavioral sciences. The studies in individual differences among programmers are one example;[15] the recent workshops on the empirical study of programmers provide another illustration.[16]

The next higher level in the medical analogy is that of the health care delivery system. Epidemiology, for example, studies the health of the population and uses domain-

specific knowledge to identify the causes of ill health, areas of potential risk, or the effects of change. The evaluation of a health care system, i.e., a system designed to alter the health status of a population, provides additional insights. In this case, the system considers benefits and costs separately. Costs are evaluated as dollar values. The benefits are organized into the following three categories:

> **Structure.** This is the capacity of the facilities, qualification of the personnel, etc. An example in the software engineering context would be the use of methods and tools that achieve the goal of "requirements analysis, design, test and maintenance of application software by technicians in an economics-driven context."[17]
>
> **Process.** This is the volume, cost, and appropriateness of activities in the achievement of the system goals. A software example here would be the impact of walkthroughs as measured by the rates of defect detection in different stages of the development process.
>
> **Outcome.** This is the change in status attributable to a system. Health-related examples would be mortality and morbidity rates. For software products, the measurable outcomes might be post-delivery error counts, evaluations of relative product performance or user satisfaction, and the ability to meet schedule or budget goals. Note that outcome measures always are relative to some baseline.

This statement of the research problem concludes with the following observation. Horizontal evaluation in software engineering just is emerging as a serious issue. There are many models to draw from in establishing evaluation methods, and the scientific quality of future research in software engineering will depend -- in part -- on how well we apply this knowledge. In the following section, the results of the investigation supported by this grant are summarized.

ACCOMPLISHMENTS

In the grant proposal, the research was divided into two categories of activity: conceptual and experimental. The goal of the conceptual tasks was to gain an understanding of evaluation methodology as it relates to software engineering. The experimental tasks, on the other hand, involved trials in the software engineering domain that would provide insight into the methodological concepts.

In late 1987 the author became the Principal Investigator of a research contract with the Office of Naval Research (ONR) to study knowledge representation in software engineering. Fortunately, this contract complemented the study under the AFOSR grants, and it was possible to enlarge and integrate the evaluation methodology investigations with the research into knowledge representation issues.

The results of some of the research conducted under this grant have been documented in the form of reviewed papers, invited presentations, and internal reports. (Because the work was organized as the second year of a three year effort, not all the results of the work initiated by this grant have been documented.) In what follows, the published results are identified. The material is grouped into two categories: those activities that were supported

by the AFOSR grant and work by the investigator that was not supported by the grant. The latter are included because they affected the PI's perceptions and in this way impacted the grant research.

Activities Supported by AFOSR Grant Only

The major accomplishment of this research is the formalization of an essential software model and the evaluation of an environment that implements this model. Briefly, the software process is described as a transformation from a need defined in the application domain to a software product that meets that need in the implementation domain. In this way, the process is best characterized as a problem solving activity.

The process can be decomposed into three transformations: from the need to a conceptual model that describes a solution, from the conceptual model to a formal model that defines the behavior of an implementation, and from that formal model to an implementation. What makes the software process so difficult is that it involves two domains, two categories of modeling tool, and three transformations.

For large projects, few can be expected to master the entire problem, and the project is decomposed into smaller, encapsulated tasks. Each such task, unfortunately, offers a limited view of the entire problem, its solution, or the process. Thus, the activities of decomposition and synthesis further compound the complexity.

Many of the insights regarding the essential software process were developed during the first year of the AFOSR-supported research. The paper Evaluating Alternative Paradigms (*Large Scale Systems*, 12:189-199, 1987) described this model and explored how it could be used to aid in evaluating alternative paradigms. The paper A Program a Day: Software Productivity's Four Minute Mile (*Proceedings, 27th Annual Technical Symposium*, Washington, D.C. Chapter of ACM, June 9, 1988, pp. 21-25) explored measures for productivity. The material in that paper then was extended to produce the paper Volume, Distance and Productivity (*Journal of Systems and Software*, 10:217-226, 1989).

The last paper observes that there is a difference between the volume of a problem to be solved and the product that realizes its solution. The smaller the volume, the less effort that the project will require, and the lower the complexity of the process. Unfortunately, reducing the volume is not a sufficient condition to improving productivity. Because the software process is based on three transformations, it also is necessary to facilitate the transformations by defining representations that reduce the conceptual distance between the models that control the transformations. That is, if the software process is a problem solving activity, then the representations used should reflect the problem being solved.

During the period of this grant, considerable work was done in expanding the understanding of these concepts. Two activities acted as driving forces for this work. First the PI was invited to prepare a paper for a special section of the *Proceedings of the IEEE* on software maintenance. The paper was entitled Improving Software Maintenance by Learning from the Past: A Case Study (*Proceedings of the IEEE*, (77,4):596-606, 1989); it was based upon an analysis of eight years of evolution with a large clinical information system (the Johns Hopkins Oncology Clinical Information System, OCIS). Although this paper focused only on issues relating to software maintenance, the analysis provided a new understanding of the software process in an unconstrained environment.

The second, and more ambitious project was the completion of a book entitled *TEDIUM and the Software Process*, which was solicited by the MIT Press. As shown in the table of contents (Appendix), the book is divided into three parts. The material in Part I establishes the basic concepts by first reviewing the software process, then building a philosophic framework for the remainder of the book, and finally describing some background information helpful in understanding what follows. Part II presents a description of TEDIUM with a major emphasis on representation issues. Part III contains two evaluations. The first is an evaluation of TEDIUM** in the context of its design objectives; the second is an evaluation of the software process in the context of the support provided by TEDIUM.

Although the above book provided the PI with an opportunity to develop his concepts without regard to the limitations of space, investigations after the completion of the manuscript have led to other papers that draw from and build on the material in the book. These are:

B. I. Blum, A Paradigm for the 1990s Validated in the 1980s, *Proc. AIAA Computers in Aerospace VII*, pp. 502-511, 1989. This paper describes the essential software model, summarizes the evaluation of TEDIUM as an instantiation of that model, and suggests how the lessons learned from this experience can be generalized to other domains.

B. I. Blum, On the Cognitive Journey to the What and How, *Proc. 28th Annual Technical Symposium of the Washington, DC Chapter of ACM*, pp. 87-94, 1989. This paper explores what happens during the transition from the problem identification to the statement of the solution and its implementation. Several student exercises are examined. Work is in progress to continue the analysis of these data for publication elsewhere.

R. Arnold, B. Blum and V. Rajlich, Bridge Technologies for Software Maintenance, *Conference on Software Maintenance*, pp. 230-231, 1989. The session addressed the need for new bridge technologies that allow the orderly transfer from old to new methods. Each participant discussed a potential approach; the PI examined the role of knowledge representation and program generation.

B. I. Blum, Prototyping and Formalism in the Software Process, invited paper for *Information and Decision Technologies*, in press. This paper explores the tension between these two approaches to software development. It suggests application domains in which each technique is the most appropriate approach.

B. I. Blum, Toward a Paperless Development Environment, *Tools for AI*, pp. 495-498, 1989. This paper identified the goal of a comprehensive environment in which the need for paper was eliminated. Methods for evaluating such an environment were presented.

B. I. Blum, A Software Environment: Some Surprising Empirical Results, *Proc. NASA/GSFC Software Engineering Workshop*, in press. This paper described some experience in the evaluation of TEDIUM data. The objective was to provide new insight into the essential software process.

** TEDIUM is a registered trademark of Tedious Enterprises, Inc.

This work was directly supported by the AFOSR grant, the ONR contract and other APL tasks.

Activities Not Supported by the AFOSR Grant

For some time the PI was an active researcher in the field of medical informatics. Although he no longer is engaged in that work, he has spent some time finishing projects begun prior to the AFOSR grant. All of this work involves domain specific application of computer technology, and consequently it augments his work on this grant. Three major activities are identified. Two represent books for which the PI was a coeditor; the third is a summary of recent invited papers and presentations.

The first of the books in this section is H. F. Orthner and B. I. Blum (eds), *Implementing Health Care Information Systems*, Springer-Verlag, New York, 1989. In addition to serving as the coeditor, the PI contributed the following chapters:

Blum, B. I. and H. F. Orthner, Implementing Health Care Information Systems, pp. 1-21.

Blum, B. I., Medical Informatics -- Phase II, pp. 22-29.

Blum, B. I., Design Methodology, pp. 277-295.

Blum, B. I., The TEDIUM Development Environment, pp. 313-352.

Blum, B. I. and H. F. Orthner, The MUMPS Programming Language, pp. 396-420.

The second of the books in this section is J. P. Enterline, R. E. Lenhard and B. I. Blum (eds), *A Clinical Information System for Oncology*, Springer-Verlag, New York, 1989. In addition to serving as coeditor, the PI contributed the following chapters:

Enterline, J. P., R. E. Lenhard and B. I. Blum, The Oncology Clinical Information System, pp. 1-21.

Blum, B. I., Development History, pp. 39-71.

Stuart, G. L., B. I. Blum and R. E. Lenhard, Clinical Data Management, pp. 73-108.

Blum, B. I., Protocol-Directed Care, pp. 109-138.

In the category of presentations, the PI was invited to deliver the following paper at a conference at the New Jersey University of Medicine and Dentistry:

Blum, B. I., Computers and Patient Care in the Nineties, *Proc. Computers in Health Sciences Symposium*, pp. 8-14, 1988.

The PI also was invited to make the following foreign presentations. (In each case, the host organization assumed responsibility for the travel and living expenses.)

Clinical Information Systems: Now and in the Future, Prince of Wales Hospital, Sydney, Australia, Sept. 7, 1988.

Issues in Clinical Information Systems (Keynote Address), Seminar on Medical Informatics and Information Management Systems, University of Newcastle, Newcastle, Australia, Sept. 9, 1988.

Systems Architecture and Software Development Methods, Seminar on Medical Informatics and Information Management Systems, University of Newcastle, Newcastle, Australia, Sept. 10, 1988.

Clinical Information Systems: Present Status and Future Potential, National University of Singapore, Singapore, Sept. 14, 1988.

Information Systems in Health Care, Seminar on Medical Informatics, Beijing Medical University, Beijing, China, October 5, 1988.

Medical Informatics, Clinical Decision Making, and Artificial Intelligence, "Smart Hospitals", Grand Fierra di Milano, Milan, Italy, April 19-22, 1989.

Considerations in Planning for Hospital Automation, University of Newcastle, Newcastle, Australia, July 18, 1989.

The Benefits of Clinical Information Systems, Grand Rounds, Prince of Wales Hospital, University of New South Wales, Sydney, Australia, July 19, 1989.

The material covered in these addresses has been documented in the following APL Research Center reports:

Blum, B. I., Medical Informatics, RMI-89-001, February, 1989.

Blum, B. I., Computer Application Architecture, RMI-89-002, February, 1989.

Blum, B. I., Medical Informatics, Clinical Decision Making, and Artificial Intelligence, RMI-89-003, March, 1989.

SUMMARY

The topic of this research involves two categories of investigation. One centers on the methods used for evaluation in the various scientific disciplines. The PI has studied these methods, but the research did not reach the point that a unifying paper directed to the software engineering problem could be produced. The second area of investigation is that of the software process and what can be evaluated with respect to it. In this domain, work progressed through small experiments and conceptual studies.

REFERENCES

1. Mathematics and Information Sciences, *Research Interests of the AFOSR*, Bolling Air Force Base, Washington, D.C. 20332, November, 1985, p. 37.

2. Musa, J. (ed.), Stimulating Software Engineering Progress, A Report of the Software Engineering Planning Group, *ACM SIGSOFT SEN*, (8,2):29-54, 1983.

3. Turski, W.M., The Role of Logic in Software Engineering, *Proceedings, 8th International Conference on Software Engineering*, IEEE Computer Society Press, 1985, p 400.

4. Sackman, H., et al., Exploratory Experimental Studies Comparing Online and Offline Programming Performance, *Communications of the ACM*, (11,1), 1968.

5. Mahoney, M.J., Self-Deception in Science, *AAAS Annual Meeting*, 1986, draft preprint, p. 1.

6. Boehm, B.W., An Experiment in Small-Scale Application Software Engineering, *IEEE Transactions on Software Engineering*, SE-7:482-493, 1981.

7. Boehm, B.W., T.E. Gray and T. Seawaldt, Prototyping vs. Specifying: A Multi-Project Experiment, *IEEE Transactions on Software Engineering*, SE-10:290-303, 1984.

8. Blum, B.I., A Paradigm for Developing Information Systems, *IEEE Transactions on Software Engineering*, SE-13:432-439, 1987.

9. Cugini, J.V., *Selection and Use of General Purpose Programming Languages* (2 Vols.), NBS Spec Pub 500-117, 1984.

10. Halstead, M., *Elements of Software Science*, Elsevier, Amsterdam, 1977.

11. Albrecht, A.J. and J.E. Gaffney, Jr., Software Function, Software Lines of Code, and Development Effort Predictions: A Software Science Validation, *IEEE Transactions on Software Engineering*, SE-8:629-648, 1983.

12. Basili, V.R., R.E. Selby and D.H. Hutchens, Experimentation in Software Engineering, *IEEE Transactions on Software Engineering*, SE-12:737-743, 1986.

13. McCabe, T., A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2:308-320, 1976.

14. Blum, B.I., *Clinical Information Systems*, Springer-Verlag, New York, NY, 1986.

15. Curtis, B., Fifteen Years of Psychology and Software Engineering: Individual Differences and Cognitive Science, *Proceedings, 7th International Conference on Software Engineering*, IEEE Computer Society Press, pp. 97-106, 1984.

16. *Empirical Studies of Programmers*, Ablex Publication, Corp., Norwood, NJ, 1986, 1987.

17. Boehm, B.W., Software Engineering, *IEEE Transactions on Computers*, C-25:1226-1241, 1976, p. 1239.

Appendix

Table of Contents for B. I. Blum, *TEDIUM and the Software Process*, MIT Press, Cambridge, MA, 1989.

Part II DESCRIPTION

Chapter 4 The Data Model

Chapter 5 Program Specifications

Chapter 6 The System